# Relational Database

a database structured to recognize relations between stored items of information.

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

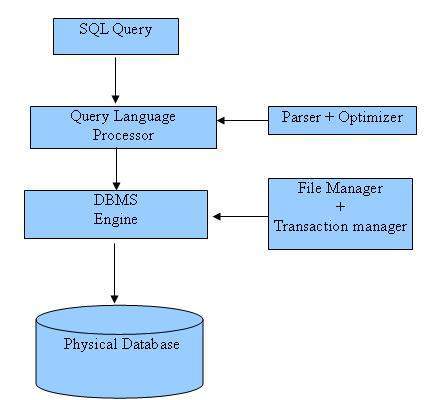
A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

## **SQL Process:**

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture:



# **Properties of a transaction**

**Atomicity:** As a transaction is set of logically related operations, **either all of them should be executed or none**. A debit transaction discussed above should either execute all three operations or none.If debit transaction fails after executing operation 1 and 2 then its new value 4000 will not be updated in the database which leads to inconsistency.

**Consistency:**If operations of debit and credit transactions on same account are executed concurrently, it may leave database in an inconsistent state.

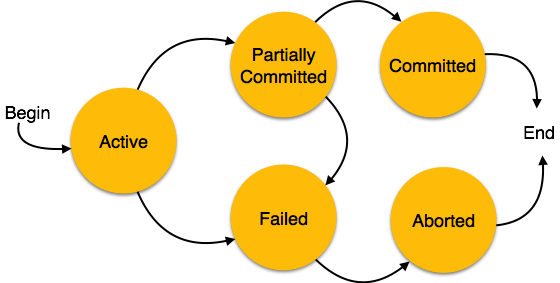
* For Example, T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches inconsistent state.
* Let us assume Account balance of A is Rs. 5000. T1 reads A(5000) and stores the value in its local buffer space. Then T2 reads A(5000) and also stores the value in its local buffer space.
* T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in T2 buffer space. T1 writes the value from its buffer back to database.
* A’s value is updated to 4000 in database and then T2 writes the value from its buffer back to database. A’s value is updated to 5500 which shows that the effect of debit transaction is lost and database has become inconsistent.
* To maintain consistency of database, we need **concurrency control protocols** which will be discussed in next article.  The operations of T1 and T2 with their buffers and database have been shown in Table 1.

**Isolation**: Result of a transaction should not be visible to others before transaction is committed. For example, Let us assume that A’s balance is Rs. 5000 and T1 debits Rs. 1000 from A. A’s new balance will be 4000. If T2 credits Rs. 500 to A’s new balance, A will become 4500 and after this T1 fails. Then we have to rollback T2 as well because it is using value produced by T1. So a transaction results are not made visible to other transactions before it commits.

**Durable**: Once database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, bank can’t say that the update has been lost. To avoid this problem, multiple copies of database are stored at different locations.

## **States of Transactions**

A transaction in a database can be in one of the following states −



* **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
* **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
* **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
* **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
  + Re-start the transaction
  + Kill the transaction
* **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

# **Concurrency Control**

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories −

* Lock based protocols
* Time stamp based protocols

## **Lock-based Protocols**

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

* **Binary Locks** − A lock on a data item can be in two states; it is either locked or unlocked.
* **Shared/exclusive** − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

Shared lock (Read lock)

Exclusive lock (Write lock)

Any number of transactions can hold shared lock on an item but exclusive lock can be held by only one transaction

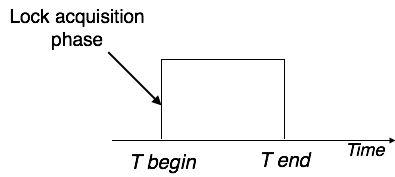
There are four types of lock protocols available −

### **Simplistic Lock Protocol**

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

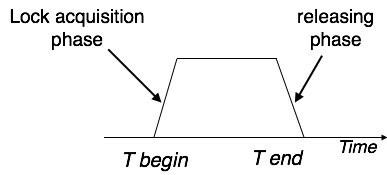
### **Pre-claiming Lock Protocol**

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



### **Two-Phase Locking 2PL**

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

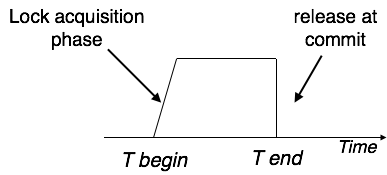


Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

### **Strict Two-Phase Locking**

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



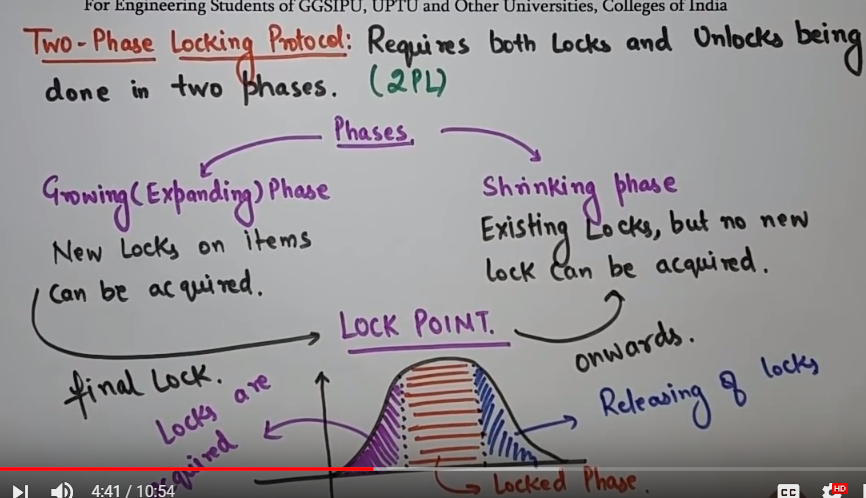
Strict-2PL does not have cascading abort as 2PL does.

**Conversion of locks:**

1. Upgrading: read lock to write lock: shared lock to exclusive lock
2. Down grading: write lock to read lock: exclusive lock to shared lock

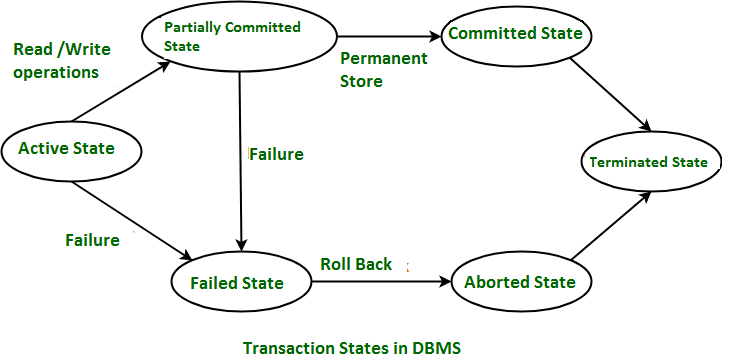
**Two Phase locking Protocol : 2PL :**

Requires both locking and unlocking being done in 2 phases.



<https://www.geeksforgeeks.org/dbms-concurrency-control-protocols-two-phase-locking-2-pl/>

## Transaction States



## Transaction Isolation Levels

<https://www.geeksforgeeks.org/transaction-isolation-levels-dbms/>

<https://fauna.com/blog/introduction-to-transaction-isolation-levels>

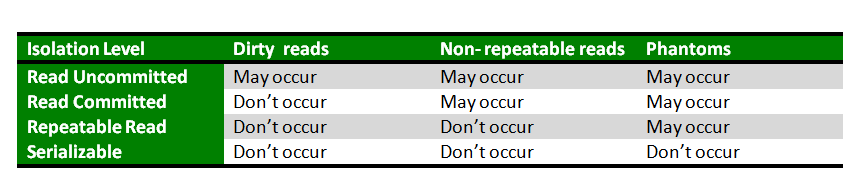
Issues with Transactions and issues to be taken care of

* **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed. For example, Let’s say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
* **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
* **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

### Isolation Levels

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The Table is given below clearly depicts the relationship between isolation levels, read phenomena and locks :



## Two Phase Commit

<https://www.geeksforgeeks.org/two-phase-commit-protocol-distributed-transaction-management/?ref=rp>

## **SQL Commands:**

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their nature:

## **DDL - Data Definition Language:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| CREATE | Creates a new table, a view of a table, or other object in database |
| ALTER | Modifies an existing database object, such as a table. |
| DROP | Deletes an entire table, a view of a table or other object in the database. |

## **DML - Data Manipulation Language:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| SELECT | Retrieves certain records from one or more tables |
| INSERT | Creates a record |
| UPDATE | Modifies records |
| DELETE | Deletes records |

## **DCL - Data Control Language:**

|  |  |
| --- | --- |
| **Command** | **Description** |
| GRANT | Gives a privilege to user |
| REVOKE | Takes back privileges granted from user |

## **What is record or row?**

A record, also called a row of data, is each individual entry that exists in a table. For example there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table:

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+----------+-----+-----------+----------+

A record is a horizontal entity in a table.

## **What is column?**

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would consist of the following:

+-----------+

| ADDRESS |

+-----------+

| Ahmedabad |

| Delhi |

| Kota |

| Mumbai |

| Bhopal |

| MP |

| Indore |

+----+------+

# Important COnstratins

* [PRIMARY Key](http://www.tutorialspoint.com/sql/sql-primary-key.htm): Uniquely identified each rows/records in a database table.
* [FOREIGN Key](http://www.tutorialspoint.com/sql/sql-foreign-key.htm): Uniquely identified a rows/records in any another database table.
* [INDEX](http://www.tutorialspoint.com/sql/sql-index.htm): Use to create and retrieve data from the database very quickly.

## **Database Normalization**

Database normalization is the process of efficiently organizing data in a database. There are two reasons of the normalization process:

* Eliminating redundant data, for example, storing the same data in more than one tables.
* Ensuring data dependencies make sense.

Normalization guidelines are divided into normal forms; think of form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database structure so that it complies with the rules of first normal form, then second normal form, and finally third normal form.

It's your choice to take it further and go to fourth normal form, fifth normal form, and so on, but generally speaking, third normal form is enough.

* [First Normal Form (1NF)](http://www.tutorialspoint.com/sql/first-normal-form.htm)
* [Second Normal Form (2NF)](http://www.tutorialspoint.com/sql/second-normal-form.htm)
* [Third Normal Form (3NF)](http://www.tutorialspoint.com/sql/third-normal-form.htm)

First normal form (1NF) sets the very basic rules for an organized database:

* Define the data items required, because they become the columns in a table. Place related data items in a table.
* Ensure that there are no repeating groups of data.
* Ensure that there is a primary key.

Second normal form states that it should meet all the rules for 1NF and there must be no partial dependences of any of the columns on the primary key.

A table is in third normal form when the following conditions are met:

* It is in second normal form.
* All nonprimary fields are dependent on the primary key.

# SQL Syntax

All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

Important point to be noted is that SQL is **case insensitive,** which means SELECT and select have same meaning in SQL statements, but MySQL makes difference in table names. So if you are working with MySQL, then you need to give table names as they exist in the database.

## **SQL SELECT Statement:**

SELECT column1, column2....columnN

FROM table\_name;

## **SQL DISTINCT Clause:**

SELECT DISTINCT column1, column2....columnN

FROM table\_name;

## **SQL WHERE Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION;

## **SQL AND/OR Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

## **SQL IN Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

## **SQL BETWEEN Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

## **SQL LIKE Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name LIKE { PATTERN };

## **SQL ORDER BY Clause:**

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

## **SQL GROUP BY Clause:**

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name;

## **SQL COUNT Clause:**

SELECT COUNT(column\_name)

FROM table\_name

WHERE CONDITION;

## **SQL HAVING Clause:**

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

## **SQL CREATE TABLE Statement:**

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

## **SQL DROP TABLE Statement:**

DROP TABLE table\_name;

## **SQL CREATE INDEX Statement :**

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

## **SQL DROP INDEX Statement :**

ALTER TABLE table\_name

DROP INDEX index\_name;

## **SQL DESC Statement :**

DESC table\_name;

## **SQL TRUNCATE TABLE Statement:**

TRUNCATE TABLE table\_name;

## **SQL ALTER TABLE Statement:**

ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype};

## **SQL ALTER TABLE Statement (Rename) :**

ALTER TABLE table\_name RENAME TO new\_table\_name;

## **SQL INSERT INTO Statement:**

INSERT INTO table\_name( column1, column2....columnN)

VALUES ( value1, value2....valueN);

## **SQL UPDATE Statement:**

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

## **SQL DELETE Statement:**

DELETE FROM table\_name

WHERE {CONDITION};

## **SQL CREATE DATABASE Statement:**

CREATE DATABASE database\_name;

## **SQL DROP DATABASE Statement:**

DROP DATABASE database\_name;

## **SQL USE Statement:**

USE database\_name;

## **SQL COMMIT Statement:**

COMMIT;

## **SQL ROLLBACK Statement:**

ROLLBACK;

# Data Types

Bigint, int, float, datetime, date, time, char, varchar, binary etc…

## **SQL Comparison Operators:**

Assume variable a holds 10 and variable b holds 20, then:

[Show Examples](http://www.tutorialspoint.com/sql/sql-comparison-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true.  Eg : SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 2000; | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

## **SQL Logical Operators:**

Here is a list of all the logical operators available in SQL.

[Show Examples](http://www.tutorialspoint.com/sql/sql-logical-operators.htm)

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | The ALL operator is used to compare a value to all values in another value set.  Eg: SQL> SELECT \* FROM CUSTOMERS WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500); |
| AND | The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.  Eg : SQL> SELECT \* FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500; |
| ANY | The ANY operator is used to compare a value to any applicable value in the list according to the condition. |
| BETWEEN | The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.  Eg: SQL> SELECT \* FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27; |
| EXISTS | The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria. |
| IN | The IN operator is used to compare a value to a list of literal values that have been specified.  Eg: SQL> SELECT \* FROM CUSTOMERS WHERE AGE IN ( 25, 27 ); |
| LIKE | The LIKE operator is used to compare a value to similar values using wildcard operators.  Eg: SQL> SELECT \* FROM CUSTOMERS WHERE NAME LIKE 'Ko%'; |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| OR | The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| IS NULL | The NULL operator is used to compare a value with a NULL value. |
| UNIQUE | The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |
| Is not null |  |

# Expressions

Syntax:

Consider the basic syntax of the SELECT statement as follows:

SELECT column1, column2, columnN FROM table\_name WHERE [CONDITION|EXPRESSION];

## **SQL - Boolean Expressions:**

SQL Boolean Expressions fetch the data on the basis of matching single value. Following is the syntax:

SELECT column1, column2, columnN

FROM table\_name

WHERE SINGLE VALUE MATCHTING EXPRESSION;

Eg:

SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 10000;

## **SQL - Numeric Expression:**

This expression is used to perform any mathematical operation in any query. Following is the syntax:

SELECT numerical\_expression as OPERATION\_NAME

[FROM table\_name

WHERE CONDITION] ;

SQL> SELECT (15 + 6) AS ADDITION

+----------+

| ADDITION |

+----------+

| 21 |

+----------+

1 row in set (0.00 sec)

SQL> SELECT COUNT(\*) AS "RECORDS" FROM CUSTOMERS;

+---------+

| RECORDS |

+---------+

| 7 |

+---------+

1 row in set (0.00 sec)

## **SQL - Date Expressions:**

Date Expressions return current system date and time values:

SQL> SELECT CURRENT\_TIMESTAMP;

+---------------------+

| Current\_Timestamp |

+---------------------+

| 2009-11-12 06:40:23 |

+---------------------+

1 row in set (0.00 sec)

# **Database**

CREATE DATABASE DatabaseName;

SHOW DATABASES;

Drop database databaseName;

Use databaseName;

# Table

## create

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

Example:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Create table from existing table:

CREATE TABLE NEW\_TABLE\_NAME AS

SELECT [ column1, column2...columnN ]

FROM EXISTING\_TABLE\_NAME

[ WHERE ]

Example:

SQL> CREATE TABLE SALARY AS

SELECT ID, SALARY

FROM CUSTOMERS;

## Drop

DROP TABLE table\_name;

## Describe

Desc tableName;

## Alter

Syntax:

add a new column in an existing table is as follows:

ALTER TABLE table\_name ADD column\_name datatype;

Drop a column

ALTER TABLE table\_name DROP COLUMN column\_name;

Change datatype:

ALTER TABLE table\_name MODIFY COLUMN column\_name datatype;

Add not null

ALTER TABLE table\_name MODIFY column\_name datatype NOT NULL;

Add unique constraint:

ALTER TABLE table\_name

ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

Add primary key:

ALTER TABLE table\_name

ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);

Drop primary key:

ALTER TABLE table\_name

DROP CONSTRAINT MyPrimaryKey;

Drop primary key For mysql:

ALTER TABLE table\_name

DROP PRIMARY KEY;

## INSERT Query

Syntax:

There are two basic syntaxes of INSERT INTO statement as follows:

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...valueN);

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQL INSERT INTO syntax would be as follows:

INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);

Populate one table using another table:

INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

## SELECT Query

SELECT column1, column2, columnN FROM table\_name;

SELECT \* FROM table\_name;

### WHERE Clause

Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition]

### AND and OR Conjunctive Operators

Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition1] AND/OR [condition2]...AND/OR [conditionN];

### Distinct

select distinct department\_id from employees;

## UPDATE Query

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

UPDATE table\_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

## DELETE Query

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

DELETE FROM table\_name

WHERE [condition];

To delete all the records:

SQL> DELETE FROM CUSTOMERS;

## LIKE Clause

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

* The percent sign (%)
* The underscore (\_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Eg :

SELECT FROM table\_name

WHERE column LIKE 'XXXX%'

## TOP, LIMIT or ROWNUM Clause

SQL> SELECT TOP 3 \* FROM CUSTOMERS;

Top works in sql server

SQL> SELECT \* FROM CUSTOMERS

LIMIT 3;

Limit works in mysql

SQL> SELECT \* FROM CUSTOMERS

WHERE ROWNUM <= 3;

Rownum works in oracle server

## ORDER BY Clause

Used to sort the data

Syntax:

The basic syntax of ORDER BY clause is as follows:

SELECT column-list

FROM table\_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

By default its asc. Asc or desc order needs to be provided by the side of every column name.

## Group By

Syntax:

SELECT column1, column2

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2

ORDER BY column1, column2

## Distinct Keyword

The SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

SELECT DISTINCT column1, column2,.....columnN

FROM table\_name

WHERE [condition]

## SORTING Results

SELECT column-list

FROM table\_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

To fetch the rows with own preferred order, the SELECT query would as follows:

SQL> SELECT \* FROM CUSTOMERS

ORDER BY (CASE ADDRESS

WHEN 'DELHI' THEN 1

WHEN 'BHOPAL' THEN 2

WHEN 'KOTA' THEN 3

WHEN 'AHMADABAD' THEN 4

WHEN 'MP' THEN 5

ELSE 100 END) ASC, ADDRESS DESC;

## Joins

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables in our SELECT statement as follows:

SQL> SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

## **SQL Join Types:**

There are different types of joins available in SQL:

* [INNER JOIN:](http://www.tutorialspoint.com/sql/sql-inner-joins.htm) returns rows when there is a match in both tables.
* [LEFT JOIN:](http://www.tutorialspoint.com/sql/sql-left-joins.htm) returns all rows from the left table, even if there are no matches in the right table.
* [RIGHT JOIN:](http://www.tutorialspoint.com/sql/sql-right-joins.htm) returns all rows from the right table, even if there are no matches in the left table.
* [FULL JOIN:](http://www.tutorialspoint.com/sql/sql-full-joins.htm) returns rows when there is a match in one of the tables.
* [SELF JOIN:](http://www.tutorialspoint.com/sql/sql-self-joins.htm) is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
* [CARTESIAN JOIN:](http://www.tutorialspoint.com/sql/sql-cartesian-joins.htm) returns the Cartesian product of the sets of records from the two or more joined tables.

### Inner Join

The most frequently used and important of the joins is the **INNER JOIN**. They are also referred to as an EQUIJOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

INNER JOIN is the same as JOIN.



Syntax:

The basic syntax of **INNER JOIN** is as follows:

SELECT table1.column1, table2.column2...

FROM table1

INNER JOIN table2

ON table1.common\_field = table2.common\_field;

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

| OID | DATE | ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using INNER JOIN as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

Eg 2:

MariaDB [htest]> select \* from employee;

+-------+------+--------+--------+

| name  | age  | salary | emp\_id |

+-------+------+--------+--------+

| name1 |   21 |  21000 |      1 |

| name2 |   21 |  22000 |      2 |

| name3 |   22 |  21000 |      3 |

| name4 |   22 |  22000 |      4 |

| name5 |   21 |  21000 |      5 |

| name6 |   21 |  21000 |      6 |

+-------+------+--------+--------+

**6 rows in set (0.00 sec)**

MariaDB [htest]> select \* from project;

+-----------+---------+

| proj\_name | proj\_id |

+-----------+---------+

| Proj1     |       1 |

| Proj2     |       2 |

| Proj3     |       3 |

| Proj4     |       4 |

+-----------+---------+

**4 rows in set (0.00 sec)**

MariaDB [htest]> select \* from emp\_proj;

+-------------+---------+--------+

| emp\_proj\_id | proj\_id | emp\_id |

+-------------+---------+--------+

|           1 |       1 |      1 |

|           2 |       1 |      2 |

|           3 |       1 |      3 |

|           4 |       2 |      3 |

|           5 |       2 |      4 |

|           6 |       3 |      5 |

|           7 |       3 |      1 |

+-------------+---------+--------+

**7 rows in set (0.00 sec)**

MariaDB [htest]>

MariaDB [htest]> select name, proj\_name from employee join emp\_proj on employee.emp\_id=emp\_proj.emp\_id join project on emp\_proj.proj\_id=project.proj\_id;

+-------+-----------+

| name  | proj\_name |

+-------+-----------+

| name1 | Proj1     |

| name2 | Proj1     |

| name3 | Proj1     |

| name3 | Proj2     |

| name4 | Proj2     |

| name5 | Proj3     |

| name1 | Proj3     |

+-------+-----------+

**7 rows in set (0.00 sec)**

### Left Join

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of **LEFT JOIN** is as follows:

SELECT table1.column1, table2.column2...

FROM table1

LEFT JOIN table2

ON table1.common\_field = table2.common\_field;

Here given condition could be any given expression based on your requirement.

In some databases LEFT JOIN is called LEFT OUTER JOIN.



Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

| OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using LEFT JOIN as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

Eg 2:

Ref eg2 of inner join.

MariaDB [htest]> select name, proj\_name from employee left join emp\_proj on employee.emp\_id=emp\_proj.emp\_id left join project on emp\_proj.proj\_id=project.proj\_id;

+-------+-----------+

| name  | proj\_name |

+-------+-----------+

| name1 | Proj1     |

| name2 | Proj1     |

| name3 | Proj1     |

| name3 | Proj2     |

| name4 | Proj2     |

| name5 | Proj3     |

| name1 | Proj3     |

| name6 | NULL      |

+-------+-----------+

**8 rows in set (0.01 sec)**

### Right Join

 In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



Eg:

Ref Eg 2 of Inner join.

MariaDB [htest]> select name, proj\_name

from employee

right join emp\_proj

on employee.emp\_id=emp\_proj.emp\_id

right join project

on emp\_proj.proj\_id=project.proj\_id;

+-------+-----------+

| name  | proj\_name |

+-------+-----------+

| name1 | Proj1     |

| name2 | Proj1     |

| name3 | Proj1     |

| name3 | Proj2     |

| name4 | Proj2     |

| name5 | Proj3     |

| name1 | Proj3     |

| NULL  | Proj4     |

+-------+-----------+

**8 rows in set (0.00 sec)**

### Full Join



## Alias Syntax

You can rename a table or a column temporarily by giving another name known as alias.

Syntax:

The basic syntax of **table** alias is as follows:

SELECT column1, column2....

FROM table\_name AS alias\_name

WHERE [condition];

## Indexes

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is as follows:

CREATE INDEX index\_name ON table\_name;

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

CREATE INDEX index\_name

ON table\_name (column\_name);

# References

Group by : <https://www.youtube.com/watch?v=FKSSOpQe5Jc>

Mysql summary : <https://www.youtube.com/watch?v=yPu6qV5byu4>